12
RESTful Web Services with Jersey and JAX-RS

Representational State Transfer (**REST**) is an architectural style in which web services are viewed as resources and can be identified by Uniform Resource Identifiers (URIs).

Web services developed using the REST style are known as RESTful web services.

Java EE 6 adds support to RESTful web services through the addition of the Java API for RESTful Web Services (JAX-RS). JAX-RS has been available as a standalone API for a while; it became part of Java EE in version 6 of the specification. In this chapter, we will cover how to develop RESTful web services through the JAX-RS API using Jersey—the JAX-RS implementation included by GlassFish.

The following topics will be covered in this chapter:

- Introduction to RESTful web services and JAX-RS
- Developing a simple RESTful web service
- Developing a RESTful web service client
- Path parameters
- Query parameters

Introduction to RESTful web services and JAX-RS

RESTful web services are very flexible. RESTful web services can consume several types of different MIME types, although they are typically written to consume and/or produce XML or JSON (JavaScript Object Notation).

Web services must support one or more of the following four HTTP methods:

- GET – By convention, a GET request is used to retrieve an existing resource
- POST – By convention, a POST request is used to update an existing resource
- PUT – By convention, a PUT request is used to create a new resource
- DELETE – By convention, a DELETE request is used to delete an existing resource

We develop a RESTful web service with JAX-RS by creating a class with annotated methods that are invoked when our web service receives one of these HTTP request methods. Once we have developed and deployed our RESTful web service, we need to develop a client that will send requests to our service. Jersey—the JAX-RS implementation included with GlassFish—includes an API that we can use to easily develop RESTful web service clients. Jersey's client-side API is a value-added feature and is not part of the JAX-RS specification.

Developing a simple RESTful web service

In this section, we will develop a simple web service to illustrate how we can make the methods in our service respond to the different HTTP request methods.

Developing a RESTful web service using JAX-RS is simple and straightforward. Each of our RESTful web services needs to be invoked via its Unique Resource Identifier (URI). This URI is specified by the `@Path` annotation, which we need to use to decorate our RESTful web service resource class.

When developing RESTful web services, we need to develop methods that will be invoked when our web service receives an HTTP request. We need to implement methods to handle one or more of the four types of requests that RESTful web services handle: GET, POST, PUT, and/or DELETE.

The JAX-RS API provides four annotations that we can use to decorate the methods in our web service. The annotations are appropriately named as `@GET`, `@POST`, `@PUT`, and `@DELETE`. Decorating a method in our web service with one of these annotations will make it respond to the corresponding HTTP method.

Additionally, each method in our service must produce and/or consume a specific MIME type. The MIME type to be produced needs to be specified with the `@Produces` annotation. Similarly, the MIME type to be consumed must be specified with the `@Consumes` annotation.

The following example illustrates the concepts we have just explained:

Please note that this example does not "really" do anything. The purpose of the example is to illustrate how to make the different methods in our RESTful web service resource class respond to the different HTTP methods.

```
package com.ensode.jaxrsintro.service;

import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("customer")
public class CustomerResource
{
  @GET
  @Produces("text/xml")
  public String getCustomer()
  {
    //in a "real" RESTful service, we would retrieve data from a
    //database
    //then return an XML representation of the data.

    System.out.println("--- " + this.getClass().getCanonicalName()
      + ".getCustomer() invoked");

    return "<customer>\n"
      + "<id>123</id>\n"
      + "<firstName>Joseph</firstName>\n"
      + "<middleName>William</middleName>\n"
      + "<lastName>Graystone</lastName>\n"
      + "</customer>\n";
  }

  /**
   * Create a new customer
   * @param customer XML representation of the customer to create
   */
```

```
    @PUT
    @Consumes("text/xml")
    public void createCustomer(String customerXML)
    {
      //in a "real" RESTful service, we would parse the XML
      //received in the customer XML parameter, then insert
      //a new row into the database.

      System.out.println("--- " + this.getClass().getCanonicalName()
        + ".createCustomer() invoked");

      System.out.println("customerXML = " + customerXML);
    }

    @POST
    @Consumes("text/xml")
    public void updateCustomer(String customerXML)
    {
      //in a "real" RESTful service, we would parse the XML
      //received in the customer XML parameter, then update
      //a row in the database.

      System.out.println("--- " + this.getClass().getCanonicalName()
        + ".updateCustomer() invoked");

      System.out.println("customerXML = " + customerXML);
    }

    @DELETE
    @Consumes("text/xml")
    public void deleteCustomer(String customerXML)
    {
      //in a "real" RESTful service, we would parse the XML
      //received in the customer XML parameter, then delete
      //a row in the database.

      System.out.println("--- " + this.getClass().getCanonicalName()
        + ".deleteCustomer() invoked");

      System.out.println("customerXML = " + customerXML);
    }
  }
```

Notice that this class is annotated with the `@Path` annotation. This annotation designates the Uniform Resource Identifier (URI) for our RESTful web service. The complete URI for our service will include the protocol, server name, port, context root, the REST resources path (see next sub-section), and the value passed to this annotation.

Assuming our web service was deployed to a server called `example.com`, using the HTTP protocol on port `8080` has a context root of `jaxrsintro`, and a REST resources path of `resources`, then the complete URI for our service would be `http://example.com:8080/jaxrsintro/resources/customer`.

As web browsers generate a GET request when pointed to a URL, we can test the GET method of our service simply by pointing the browser to our service's URI.

Notice that each of the methods in our class is annotated with one of the `@GET`, `@POST`, `@PUT`, or `@DELETE` annotations. These annotations make our methods respond to the corresponding HTTP method.

Additionally, if our method returns data to the client, we declare the MIME type of the data to be returned in the `@Produces` annotation. In our example, only the `getCustomer()` method returns data to the client. We wish to return data in XML format, therefore, we set the value of the `@Produces` annotation to `"text/xml"`. Similarly, if our method needs to consume data from the client, we need to specify the MIME type of the data to be consumed. This is done via the `@Consumes` annotation. All methods except `getCustomer()` in our service consume data. In all cases, we expect the data to be in XML, therefore, we again specify `"text/xml"` as the MIME type to be consumed.

Configuring the REST resources path for our application

As briefly mentioned in the previous section, before successfully deploying a RESTful web service developed using JAX-RS, we need to configure the REST resources path for our application. There are two ways of doing this: we can either use the `web.xml` deployment descriptor or develop a class that extends `javax.ws.rs.core.Application` and decorate it with the `@ApplicationPath` annotation.

Configuring via web.xml

We can configure the REST resources path for our JAX-RS RESTful web services via the `web.xml` deployment descriptor.

Jersey libraries include a servlet that we can configure as usual in our `web.xml` deployment descriptor.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">
  <servlet>
    <servlet-name>JerseyServlet</servlet-name>
    <servlet-class>
      com.sun.jersey.spi.container.servlet.ServletContainer
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>JerseyServlet</servlet-name>
    <url-pattern>/resources/*</url-pattern>
  </servlet-mapping>
</web-app>
```

As we can see in this markup, the fully qualified class name of the Jersey servlet is `com.sun.jersey.spi.container.servlet.ServletContainer`, which is the value we add to the `<servlet-class>` element in `web.xml`. We then give this servlet a logical name (we chose `JerseyServlet` in our example), then declare the URL pattern to be handled by the servlet as usual. Any URLs matching the pattern will be directed to the appropriate method in our RESTful web services.

As we can see, configuring the Jersey servlet isn't any different from configuring any other servlet.

Configuring via the @ApplicationPath annotation

As mentioned in previous chapters, Java EE 6 adds several new features to the Java EE specification, so that in many cases, it isn't necessary to write a `web.xml` deployment descriptor. JAX-RS is no different; we can configure the REST resources path in Java code via an annotation.

To configure our REST resources path without having to rely on a `web.xml` deployment descriptor, all we need to do is write a class that extends `javax.ws.ApplicationPath` and decorate it with the `@ApplicationPath` annotation. The value passed to this annotation is the REST resources path for our services.

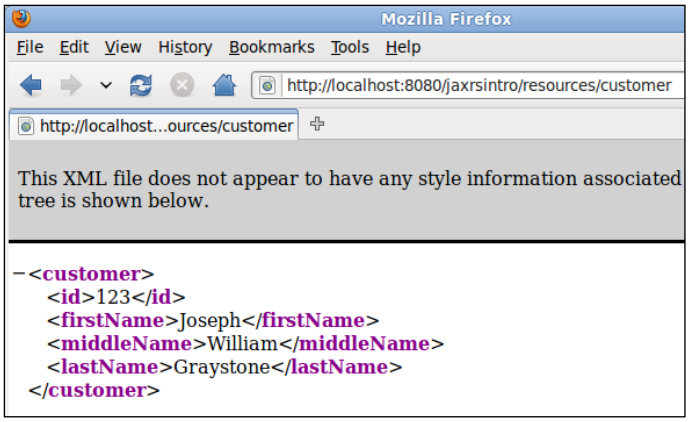
The following code sample illustrates this process:

```
package com.ensode.jaxrsintro.service.config;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("resources")
public class JaxRsConfig extends Application
{
}
```

Notice that the class does not have to implement any methods. It simply needs to extend `javax.ws.rs.Application` and get decorated with the `@ApplicationPath` annotation. The class must be public, may have any name, and may be placed in any package.

Testing our web service

Like we mentioned earlier, web browsers send a GET request to any URLs we point them to. Therefore, the easiest way to test GET requests to our service is to simply point the browser to our service's URI.



Web browsers only support GET and POST requests. To test a POST request through the browser, we would have to write a web application containing an HTML form having an action attribute value of our service's URI. Although trivial for a single service, it can become cumbersome to do this for every RESTful web service we develop.

Thankfully, there is an open source command line utility called `curl`, which we can use to test our web services. `curl` is included with most Linux distributions and can be easily downloaded for Windows, Mac OS X, and several other platforms. `curl` can be downloaded from `http://curl.haxx.se/`.

`curl` can send all four request method types (GET, POST, PUT, and DELETE) to our service. Our server's response will simply be displayed on the command line console. `curl` takes a `-X` command line option that allows us to specify what request method to send. To send a GET request, we simply need to type the following command into the command line:

```
curl -XGET http://localhost:8080/jaxrsintro/resources/customer
```

This results in the following output:

```
<customer>
<id>123</id>
<firstName>Joseph</firstName>
<middleName>William</middleName>
<lastName>Graystone</lastName>
</customer>
```

This, unsurprisingly, is the same output we saw when we pointed our browser to our service's URI.

The default request method for `curl` is GET. Therefore, the `-X` parameter in our previous example is redundant. We could have achieved the same result by invoking the following command from the command line:

```
curl http://localhost:8080/jaxrsintro/resources/customer
```

After submitting any of the previous two commands and examining the GlassFish log, we should see the output of the `System.out.println()` statements we added to the `getCustomer()` method:

```
INFO: --- com.ensode.jaxrsintro.service.CustomerResource.getCustomer()
invoked
```

For all other request method types, we need to send some data to our service. This can be accomplished by the `--data` command line argument to `curl`:

```
curl -XPUT -HContent-type:text/xml --data
"<customer><id>321</id><firstName>Amanda</firstName><middleName>Zoe
</middleName><lastName>Adams</lastName></customer>"
http://localhost:8080/jaxrsintro/resources/customer
```

As can be seen in this example, we need to specify the MIME type via the `-H` command line argument in `curl` using the format seen in the example.

We can verify that the previous command worked as expected by inspecting the GlassFish log:

```
INFO: ---
com.ensode.jaxrsintro.service.CustomerResource.createCustomer() invoked
INFO: customerXML =
<customer><id>321</id><firstName>Amanda</firstName><middleName>Zoe
</middleName><lastName>Adams</lastName></customer>
```

We can test other request method types just as easily:

```
curl -XPOST -HContent-type:text/xml --data
"<customer><id>321</id><firstName>Amanda</firstName><middleName>Tamara
</middleName><lastName>Adams</lastName></customer>"
http://localhost:8080/jaxrsintro/resources/customer
```

This results in the following output in the GlassFish log:

```
INFO: ---
com.ensode.jaxrsintro.service.CustomerResource.updateCustomer() invoked
INFO: customerXML =
<customer><id>321</id><firstName>Amanda</firstName><middleName>Tamara
</middleName><lastName>Adams</lastName></customer>
```

We can test the delete method by executing the following command:

```
curl -XDELETE -HContent-type:text/xml --data
"<customer><id>321</id><firstName>Amanda</firstName><middleName>Tamara
</middleName><lastName>Adams</lastName></customer>"
http://localhost:8080/jaxrsintro/resources/customer
```

This results in the following output in the GlassFish log:

```
INFO: ---
com.ensode.jaxrsintro.service.CustomerResource.deleteCustomer() invoked
INFO: customerXML =
<customer><id>321</id><firstName>Amanda</firstName><middleName>Tamara
</middleName><lastName>Adams</lastName></customer>
```

Converting data between Java and XML with JAXB

In our previous example, we were processing "raw" XML received as a parameter, as well as returning "raw" XML to our client. In a real application, we would more than likely parse the XML received from the client and use it to populate a Java object. Additionally, any XML that we need to return to the client would have to be constructed from a Java object.

Converting data from Java to XML and back is such a common use case that the Java EE specification provides an API to do it. This API is the Java API for XML Binding (JAXB).

JAXB makes converting data from Java to XML transparent and trivial. All we need to do is decorate the class we wish to convert to XML with the `@XmlRootElement` annotation. The following code example illustrates how to do this:

```
package com.ensode.jaxrstest.entity;

import java.io.Serializable;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Customer implements Serializable
{
  private Long id;
  private String firstName;
  private String middleName;
  private String lastName;

  public Customer()
  {
  }

  public Customer(Long id, String firstName, String middleInitial,
    String lastName)
  {
    this.id = id;
    this.firstName = firstName;
    this.middleName = middleInitial;
    this.lastName = lastName;
  }

  public String getFirstName()
  {
    return firstName;
```

```
    }

    public void setFirstName(String firstName)
    {
      this.firstName = firstName;
    }

    public Long getId()
    {
      return id;
    }

    public void setId(Long id)
    {
      this.id = id;
    }

    public String getLastName()
    {
      return lastName;
    }

    public void setLastName(String lastName)
    {
      this.lastName = lastName;
    }

    public String getMiddleName()
    {
      return middleName;
    }

    public void setMiddleName(String middleName)
    {
      this.middleName = middleName;
    }

    @Override
    public String toString()
    {
      return "id = " + getId() + "\nfirstName = " + getFirstName()
        + "\nmiddleName = " + getMiddleName() + "\nlastName = "
        + getLastName();
    }
  }
```

As we can see, other than the `@XmlRootElement` annotation at the class level, there is nothing unusual about this Java class.

Once we have a class that we have decorated with the `@XmlRootElement` annotation, we need to change the parameter type of our web service from `String` to our custom class:

```
package com.ensode.jaxbxmlconversion.service;

import com.ensode.jaxbxmlconversion.entity.Customer;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("customer")
public class CustomerResource
{
  private Customer customer;

  public CustomerResource()
  {
    //"fake" the data, in a real application the data
    //would come from a database.
    customer = new Customer(1L, "David", "Raymond", "Heffelfinger");
  }

  @GET
  @Produces("text/xml")
  public Customer getCustomer()
  {
    //in a "real" RESTful service, we would retrieve data from a
    //database
    //then return an XML representation of the data.

    System.out.println("--- " + this.getClass().getCanonicalName()
      + ".getCustomer() invoked");

    return customer;
  }

  @POST
```

```
  @Consumes("text/xml")
  public void updateCustomer(Customer customer)
  {
    //in a "real" RESTful service, JAXB would parse the XML
    //received in the customer XML parameter, then update
    //a row in the database.

    System.out.println("--- " + this.getClass().getCanonicalName()
      + ".updateCustomer() invoked");
    System.out.println("---- got the following customer: "
      + customer);
  }

  @PUT
  @Consumes("text/xml")
  public void createCustomer(Customer customer)
  {
    //in a "real" RESTful service, we would insert
    //a new row into the database with the data in th    //customer
parameter

    System.out.println("--- " + this.getClass().getCanonicalName()
      + ".createCustomer() invoked");
    System.out.println("customer = " + customer);
  }

  @DELETE
  @Consumes("text/xml")
  public void deleteCustomer(Customer customer)
  {
    //in a "real" RESTful service, we would delete a row
    //from the database corresponding to the customer parameter

    System.out.println("--- " + this.getClass().getCanonicalName()
      + ".deleteCustomer() invoked");
    System.out.println("customer = " + customer);
  }
}
```

As we can see, the difference between this version of our RESTful web service and the previous one is that all parameter types and return values have been changed from `String` to `Customer`. JAXB takes care of converting our parameters and return types to and from XML as appropriate. When using JAXB, an object of our custom class is automatically populated with data from the XML sent from the client. Similarly, return values are transparently converted to XML.

Developing a RESTful web service client

Although `curl` allows us to quickly test our RESTful web services and it is a developer-friendly tool, it is not exactly user friendly. We shouldn't expect to have our user enter `curl` commands in their command line to use our web service. For this reason, we need to develop a client for our services. Jersey—the JAX-RS implementation included with GlassFish—includes a client-side API that we can use to easily develop client applications.

The following example illustrates how to use the Jersey client API:

```
package com.ensode.jaxrsintroclient;

import com.ensode.jaxbxmlconversion.entity.Customer;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.UniformInterface;
import com.sun.jersey.api.client.WebResource;
import javax.ws.rs.core.MediaType;

public class App
{
  private WebResource baseUriWebResource;
  private WebResource webResource;
  private Client client;
  private static final String BASE_URI =
    "http://localhost:8080/jaxbxmlconversion/resources";

  public static void main(String[] args)
  {
    App app = new App();
    app.initWebResource();
    app.getCustomer();
    app.insertCustomer();
  }

  private void initWebResource()
  {
    com.sun.jersey.api.client.config.ClientConfig config =
      new com.sun.jersey.api.client.config.DefaultClientConfig();
    client = Client.create(config);
    baseUriWebResource = client.resource(BASE_URI);
    webResource = baseUriWebResource.path("customer");
  }

  public void getCustomer()
```

```
    {
      UniformInterface uniformInterface =
        webResource.type(MediaType.TEXT_XML);
      Customer customer = uniformInterface.get(Customer.class);
      System.out.println("customer = " + customer);
    }

    public void insertCustomer()
    {
      Customer customer = new Customer(234L, "Tamara", "A",
        "Graystone");
      UniformInterface uniformInterface =
        webResource.type(MediaType.TEXT_XML);
      uniformInterface.put(customer);
    }
  }
```

The first thing we need to do is create an instance of `com.sun.jersey.api.client.config.DefaultClientConfig`, then pass it to the static `create()` method of the `com.sun.jersey.api.client.Client` class. At this point, we have created an instance of `com.sun.jersey.api.client.Client`. We then need to create an instance of `com.sun.jersey.api.client.WebResource` by invoking the `resource()` method of our newly created `Client` instance, passing the base URI of our web service, as defined in its configuration, as explained earlier in this chapter.

Once we have a `WebResource` instance pointing to the base URI of our web service, we need to create a new instance pointing to the URI of the specific web service we need to target, as defined in its `@Path` annotation. We can do this simply by invoking the `path()` method on `WebResource` and passing a value matching the contents of the `@Path` annotation of our RESTful web service.

The `WebResource` class has a `type()` method that returns an instance of a class implementing `com.sun.jersey.api.client.UniformInterface`. The `type()` method takes a `String` as a parameter that can be used to indicate the MIME type that the web service will handle. The `javax.ws.rs.core.MediaType` class has several `String` constants defined, corresponding to most supported MIME types. In our example, we have been using XML, therefore, we used the corresponding `MediaType.TEXT_XML` constant as the value for this method.

The `UniformInterface` has methods we can invoke to generate the GET, POST, PUT, and DELETE HTTP requests. These methods are appropriately named as `get()`, `post()`, `put()`, and `delete()`.

In the `getCustomer()` method in our example, we invoke the `get()` method that generates a GET request to our web service. Notice that we pass the Java class of the type of data we expect to receive. JAXB automatically populates an instance of this class with the data returned from the web service.

In the `insertCustomer()` method in our example, we invoke the `put()` method on the `UniformInterface` implementation returned by `WebResource.type()`. We pass an instance of our `Customer` class, which JAXB automatically converts to XML before sending it to the server. The same technique can be used when invoking the `post()` and `delete()` methods of `UniformInterface`.

Query and path parameters

In our previous example, we have been working with a RESTful web service to manage a single customer object. In real life, this would obviously not be very helpful. A common case is to develop a RESTful web service to handle a collection of objects (in our example, customers). To determine what specific object in the collection we are working with, we can pass parameters to our RESTful web services. There are two types of parameters we can use: query and path parameters.

Query parameters

We can add parameters to methods that will handle HTTP requests in our web service. Parameters decorated with the `@QueryParam` annotation will be retrieved from the request URL.

The following example illustrates how to use query parameters in our JAX-RS RESTful web services:

```
package com.ensode.queryparams.service;

import com.ensode.queryparams.entity.Customer;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;

@Path("customer")
public class CustomerResource
```

```
{
  private Customer customer;

  public CustomerResource()
  {
    customer = new Customer(1L, "Samuel", "Joseph", "Willow");
  }

  @GET
  @Produces("text/xml")
  public Customer getCustomer(@QueryParam("id") Long id)
  {
    //in a "real" RESTful service, we would retrieve data from a
    //database
    //using the supplied id.

    System.out.println("--- " + this.getClass().getCanonicalName()
      + ".getCustomer() invoked, id = " + id);
    return customer;
  }

  /**
   * Create a new customer
   * @param customer XML representation of the customer to create
   */
  @PUT
  @Consumes("text/xml")
  public void createCustomer(Customer customer)
  {
    //in a "real" RESTful service, we would parse the XML
    //received in the customer XML parameter, then insert
    //a new row into the database.

    System.out.println("--- " + this.getClass().getCanonicalName()
      + ".createCustomer() invoked");
    System.out.println("customer = " + customer);
  }

  @POST
  @Consumes("text/xml")
  public void updateCustomer(Customer customer)
  {
    //in a "real" RESTful service, we would parse the XML
    //received in the customer XML parameter, then update
    //a row in the database.
```

```
      System.out.println("--- " + this.getClass().getCanonicalName()
        + ".updateCustomer() invoked");
      System.out.println("customer = " + customer);
      System.out.println("customer= " + customer);
    }

    @DELETE
    @Consumes("text/xml")
    public void deleteCustomer(@QueryParam("id") Long id)
    {
      //in a "real" RESTful service, we would invoke
      //a DAO and delete the row in the database with the
      //primary key passed as the "id" parameter.

      System.out.println("--- " + this.getClass().getCanonicalName()
        + ".deleteCustomer() invoked, id = " + id);
      System.out.println("customer = " + customer);
    }
  }
```

Notice that all we had to do was decorate the parameters with the `@QueryParam` annotation. This annotation allows JAX-RS to retrieve any query parameters matching the value of the annotation and assign its value to the parameter variable.

We can add a parameter to the web service's URL, just like we pass parameters to any URL:

```
curl -XGET -HContent-type:text/xml
http://localhost:8080/queryparams/resources/customer?id=1
```

Sending query parameters via the Jersey client API

The Jersey client API provides an easy and straightforward way of sending query parameters to RESTful web services. The following example illustrates how to do this:

```
package com.ensode.queryparamsclient;

import com.ensode.queryparamsclient.entity.Customer;
import com.sun.jersey.api.client.Client;
import com.sun.jersey.api.client.UniformInterface;
import com.sun.jersey.api.client.WebResource;
import javax.ws.rs.core.MediaType;

public class App
{
  private WebResource baseUriWebResource;
```

```
    private WebResource webResource;
    private Client client;
    private static final String BASE_URI =
      "http://localhost:8080/queryparams/resources";

    public static void main(String[] args)
    {
      App app = new App();
      app.initWebResource();
      app.getCustomer();
    }

    private void initWebResource()
    {
      com.sun.jersey.api.client.config.ClientConfig config =
        new com.sun.jersey.api.client.config.DefaultClientConfig();
      client = Client.create(config);
      baseUriWebResource = client.resource(BASE_URI);
      webResource = baseUriWebResource.path("customer");
    }

    public void getCustomer()
    {
      UniformInterface uniformInterface =
        webResource.type(MediaType.TEXT_XML);

      Customer customer =
     (Customer) webResource.queryParam("id", "1").get( Customer.class);
      System.out.println("customer = " + customer);
    }
  }
```

As we can see, all we need to do to pass a parameter is to invoke the `queryParam()` method on `com.sun.jersey.api.client.WebResource`. The first argument to this method is the parameter name and it must match the value of the `@QueryParam` annotation on the web service. The second parameter is the value we need to pass to the web service.

If we need to pass multiple parameters to one of our web service's methods, then we need to use an instance of a class implementing the `javax.ws.rs.core.MultiValuedMap` interface. Jersey provides a default implementation in the form of `com.sun.jersey.core.util.MultiValuedMapImpl` that should suffice for most cases.

The following code fragment illustrates how to pass multiple parameters to a web service method:

```
MultivaluedMap multivaluedMap = new MultivaluedMapImpl();
multivaluedMap.add("paramName1", "value1");
multivaluedMap.add("paramName2", "value2");
String s = webResource.queryParams(multivaluedMap).get(String.class);
```

We need to add all the parameters we need to send to our web service by invoking the `add()` method on our `MultivaluedMap` implementation. This method takes the parameter name as its first argument and the parameter value as its second argument. We need to invoke this method for each parameter we need to send.

As we can see, `com.sun.jersey.api.client.WebResource` has a `queryParams()` method that takes an instance of a class implementing the `MultivaluedMap` interface as a parameter. In order to send multiple parameters to our web service, we simply need to pass an instance of `MultivaluedMap` containing all required parameters to this method.

Path parameters

Another way by which we can pass parameters to our RESTful web services is via path parameters. The following example illustrates how to develop a JAX-RS RESTful web service that accepts path parameters:

```
package com.ensode.pathparams.service;

import com.ensode.pathparams.entity.Customer;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

@Path("/customer/")
public class CustomerResource
{
  private Customer customer;

  public CustomerResource()
  {
    customer = new Customer(1L, "William", "Daniel", "Graystone");
```

```
}

@GET
@Produces("text/xml")
@Path("{id}/")
public Customer getCustomer(@PathParam("id") Long id)
{
  //in a "real" RESTful service, we would retrieve data from a
  //database
  //using the supplied id.

  System.out.println("--- " + this.getClass().getCanonicalName()
    + ".getCustomer() invoked, id = " + id);
  return customer;
}

@PUT
@Consumes("text/xml")
public void createCustomer(Customer customer)
{
  //in a "real" RESTful service, we would parse the XML
  //received in the customer XML parameter, then insert
  //a new row into the database.

  System.out.println("--- " + this.getClass().getCanonicalName()
    + ".createCustomer() invoked");
  System.out.println("customer = " + customer);
}

@POST
@Consumes("text/xml")
public void updateCustomer(Customer customer)
{
  //in a "real" RESTful service, we would parse the XML
  //received in the customer XML parameter, then update
  //a row in the database.

  System.out.println("--- " + this.getClass().getCanonicalName()
    + ".updateCustomer() invoked");
  System.out.println("customer = " + customer);
  System.out.println("customer= " + customer);
}

@DELETE
```

```
    @Consumes("text/xml")
    @Path("{id}/")
    public void deleteCustomer(@PathParam("id") Long id)
    {
      //in a "real" RESTful service, we would invoke
      //a DAO and delete the row in the database with the
      //primary key passed as the "id" parameter.

      System.out.println("--- " + this.getClass().getCanonicalName()
        + ".deleteCustomer() invoked, id = " + id);
      System.out.println("customer = " + customer);
    }
  }
```

Any method that accepts a path parameter must be decorated with the `@Path` annotation. The value attribute of this annotation must be formatted as `"{paramName}/"`, where `paramName` is the parameter the method expects to receive. Additionally, method parameters must be decorated with the `@PathParam` annotation. The value of this annotation must match the parameter name declared in the `@Path` annotation for the method.

We can pass path parameters from the command line by adjusting our web service's URI as appropriate. For example, to pass an "id" parameter of 1 to the previous `getCustomer()` method (which handles HTTP GET requests), we could do it from the command line as follows:

```
curl -XGET -HContent-type:text/xml
http://localhost:8080/pathparams/resources/customer/1
```

This returns the expected output of an XML representation of the `Customer` object returned by the `getCustomer()` method:

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?><customer><firstName>William</firstName><id>1</id>
<lastName>Graystone</lastName><middleName>Daniel</middleName></customer>
```

Sending path parameters via the Jersey client API

Sending path parameters to a web service via the Jersey client API is easy and straightforward; all we need to do is append any path parameters to the path we use to create our `WebResource` instance. The following example illustrates how to do this:

```
package com.ensode.queryparamsclient;

import com.ensode.queryparamsclient.entity.Customer;
import com.sun.jersey.api.client.Client;
```

```
import com.sun.jersey.api.client.WebResource;

public class App
{
  private WebResource baseUriWebResource;
  private WebResource webResource;
  private Client client;
  private static final String BASE_URI =
    "http://localhost:8080/queryparams/resources";

  public static void main(String[] args)
  {
    App app = new App();
    app.initWebResource();
    app.getCustomer();
  }

  private void initWebResource()
  {
    com.sun.jersey.api.client.config.ClientConfig config =
      new com.sun.jersey.api.client.config.DefaultClientConfig();
    client = Client.create(config);
    baseUriWebResource = client.resource(BASE_URI);
    webResource = baseUriWebResource.path("customer/1");
  }

  public void getCustomer()
  {
    Customer customer =
      (Customer) webResource.get(Customer.class);
    System.out.println("customer = " + customer);
  }
}
```

In this example, we simply appended a value of `1` as the path parameter to the `String` used to build the `WebResource` implementation used to invoke our web service. This parameter is automatically picked up by the JAX-RS API and assigned to the method argument annotated with the corresponding `@PathParam` annotation.

If we need to pass more than one parameter to one of our web services, we simply need to use the following format for the `@Path` parameter at the method level:

```
@Path("/{paramName1}/{paramName2}/")
```

Then, annotate the corresponding method arguments with the `@PathParam` annotation:

```
public String someMethod(@PathParam("paramName1") String param1,
  @PathParam("paramName2") String param2)
```

The web service can then be invoked by modifying the web service's URI to pass the parameters in the order specified in the `@Path` annotation. For example, the following URI would pass the values `1` and `2` for `paramName1` and `paramName2`:

```
http://localhost:8080/contextroot/resources/customer/1/2
```

This URI will work both from the command line or through a web service client we develop with the Jersey client API.

Summary

In this chapter, we discussed how to easily develop RESTful web services using JAX-RS—a new addition to the Java EE specification.

We covered how to develop a RESTful web service by adding a few simple annotations to our code. We also explained how to automatically convert data between Java and XML by taking advantage of the Java API for XML Binding (JAXB).

Finally, we covered how to pass parameters to our RESTful web services via the `@PathParam` and `@QueryParam` annotations.